

Unit Testing in Java

▼ Setting Up Junit in Java

- JUnit can be added as a dependency in your project using Maven or Gradle.
- **JUnit Jupiter** (JUnit 5) introduces new features and better support for Java 8+.

Maven Dependency Example:

For running JUnit 5:

Running Tests in an IDE (Eclipse/IntelliJ IDEA):

Right-click on the test class or method and select "Run as JUnit Test."

▼ Testing Edge Cases

- Edge cases occur at the boundary limits of inputs, such as the minimum, maximum, or unusual inputs that the system might handle.
- They help uncover issues that wouldn't be revealed by testing typical use cases or "happy paths."

▼ Examples of Edge Cases:

- **Empty or Null Inputs:** Functions that handle collections or strings should be tested to ensure they behave correctly when provided with empty or null inputs.
- **Boundary Values:** Testing the extremes of input values, such as the maximum or minimum limits for integers, strings, or arrays, helps ensure proper handling of these conditions.
- One-Off Errors (off-by-one): In loops or iterative processes, it's common to introduce off-by-one errors. Testing with just one item or the last item in a collection helps uncover these mistakes.
- Large Inputs: Test how the code behaves with very large datasets or numbers. This is especially important for performance and memory management.

▼ Example: Testing Edge Cases in Java

Consider a method that calculates the factorial of a number:

```
public class Calculator {
   public int factorial(int n) {
```

```
if (n < 0) throw new IllegalArgumentException("Inpu
if (n == 0 || n == 1) return 1;
int result = 1;
for (int i = 2; i <= n; i++) {
    result *= i;
}
return result;
}</pre>
```

Edge Case Tests for Factorial:

```
@Test
public void testFactorialEdgeCases() {
    Calculator calc = new Calculator();

    // Test lower boundary (edge case: 0)
    assertEquals(1, calc.factorial(0), "Factorial of 0 shou

    // Test edge case: 1
    assertEquals(1, calc.factorial(1), "Factorial of 1 shou

    // Test negative input (invalid case)
    assertThrows(IllegalArgumentException.class, () -> calc

    // Test large input (performance and correctness edge cassertEquals(120, calc.factorial(5), "Factorial of 5 sh
}
```

Importance of Edge Case Testing:

- Prevents Failures in Rare Situations: While normal inputs might work as expected, edge cases reveal potential system vulnerabilities.
- Validates Code's Robustness: Edge case testing forces you to consider rare but possible scenarios that could result in bugs.

▼ Exception Handling Testing

• Testing for exceptions ensures that your program handles errors gracefully without crashing.

• This includes testing both **expected** exceptions (ones that should occur in certain scenarios) and **unexpected** exceptions (bugs or flaws in logic).

▼ Types of Exceptions to Test For:

- IllegalArgumentException: When invalid arguments are passed to a method (e.g., null values, out-of-bounds inputs).
- **Checked Exceptions:** These are exceptions that a method is expected to handle, such as **IDEXCEPTION** Or **SQLEXCEPTION**.
- **Custom Exceptions:** Many applications define custom exceptions to handle specific error conditions. Unit tests should ensure these are thrown and handled correctly.

▼ Example: Exception Testing in Java

Let's modify the factorial function to handle the case where the input is negative:

```
@Test
public void testFactorialForExceptions() {
    Calculator calc = new Calculator();

    // Testing exception for negative input
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        calc.factorial(-5);
    });

    String expectedMessage = "Input must be non-negative";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}
```

▼ Guidelines for Exception Testing:

• Test Expected Exceptions:

Ensure that functions handle incorrect inputs by throwing the correct exceptions (e.g.,

illegalArgumentException for invalid arguments).

Test Unintended Exceptions:

Make sure that functions don't throw unintended exceptions like NullPointerException unless expected, especially when inputs are optional or default values are expected.

▼ Parameterised Testing in Junit

 Parameterised Tests allow us to run the same test with different sets of inputs, helping to reduce duplication.

Example:

```
@ParameterizedTest
@ValueSource(ints = { 2, 4, 6, 8 })
public void testEvenNumbers(int number) {
    assertTrue(number % 2 == 0, "Number should be even");
}
```

• The test will automatically run multiple times, each time with a different value from the ValueSource.

Use Case:

 Parameterised tests are especially useful when testing a function that behaves similarly across multiple inputs, such as checking if numbers are even or odd.

▼ Isolating Dependencies with Mockito

 When writing unit tests, you often need to isolate your code from dependencies like databases or external APIs. Mockito allows you to mock these dependencies.

Basic Mockito Example:

```
@Test
public void testGetMovieDetails() {
    MovieRepository mockRepo = mock(MovieRepository.clas
s);
    when(mockRepo.findById(1L)).thenReturn(Optional.of(ne
w Movie("Inception", "Sci-Fi", 2010)));

    MovieService service = new MovieService(mockRepo);
    Movie movie = service.getMovieDetails(1L);
```

```
assertEquals("Inception", movie.getTitle());
}
```

• In this example, the MovieRepository is mocked so the actual database isn't involved. We simulate the response when findById() is called.

▼ Best Practices in Unit Testing

- Keep Tests Small and Focused: A test should focus on one behavior or scenario.
- **Descriptive Test Names**: Test method names should clearly describe the functionality they are validating.

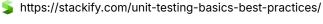
```
// Good
@Test
public void shouldReturnCorrectTotalWhenTwoItemsAreAdded
() { ... }

// Bad
@Test
public void testCart() { ... }
```

- **Test Both Happy Paths and Edge Cases**: Ensure you cover a range of inputs, including invalid or boundary values.
- Don't Overuse Mocks: While mocking is useful, too much mocking can obscure real issues.

Unit Testing Tutorial: 6 Best Practices to Get Up To Speed

Plenty of developers still do not have exposure to the unit testing practice. Read our unit testing basics best practices.





▼ Integrating Unit Tests in CI/CD

• Automated Unit Tests: Unit tests are automatically run in CI/CD pipelines, providing immediate feedback on the code quality.

Steps to Integrate Unit Testing in CI/CD:

1. Ensure all unit tests run automatically during builds.

2. Fail the build if any test fails, ensuring only high-quality code moves to production.

Code Example for GitHub Actions:

```
# This workflow will build a Java project with Maven, and
cache/restore any dependencies to improve the workflow ex
ecution time
# For more information see: https://docs.github.com/en/ac
tions/automating-builds-and-tests/building-and-testing-ja
va-with-maven
# This workflow uses actions that are not certified by Gi
tHub.
# They are provided by a third-party and are governed by
# separate terms of service, privacy policy, and support
# documentation.
name: Java CI with Maven
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - name: Set up JDK 17
      uses: actions/setup-java@v4
      with:
        java-version: '17'
        distribution: 'temurin'
        cache: maven
```

- name: Build with Maven run: mvn clean verify